MTIMESX

Fast Matrix Multiply for MATLAB®
With Multi-Dimensional Support

Version 1.11

January 6, 2010

By James Tursa

1) Introduction

mtimesx is a fast general purpose matrix and scalar multiply routine that utilizes BLAS calls and custom code to perform the calculations. **mtimesx** also has extended support for n-Dimensional (nD, n > 2) arrays, treating these as arrays of 2D matrices for the purposes of matrix operations. BLAS stands for Basic Linear Algebra Subroutines. The BLAS is a library of highly optimized routines for various vector-vector, matrix-vector, and matrix-matrix linear algebra operations. MATLAB makes calls to their BLAS library in the background whenever you do a matrix multiply. **mtimesx** links to and calls these same BLAS library routines directly.

"Doesn't MATLAB already do this?" For 2D matrices, yes, it does. However, MATLAB does not always implement the most efficient algorithms for memory access, and MATLAB does not always take full advantage of symmetric and conjugate cases. The **mtimesx** 'SPEED' mode attempts to do both of these to the fullest extent possible, and in some cases can outperform MATLAB by 300% - 400% for speed (yes, you read that right, 3x – 4x faster). For nD matrices (treating them as arrays of 2D matrices), MATLAB does not have direct support for this. One is forced to write loops to accomplish the same thing that **mtimesx** can do faster. NOTE: The MATLAB intrinsic function for matrix multiplication is called mtimes. i.e., when you type the expression A * B, MATLAB actually calls the function mtimes(A,B). In all of the discussions below, mtimes (without the x) always refers to the MATLAB built-in matrix multiply operation.

2) Operating Modes

mtimesx has two operating modes:

'MATLAB' mode

This mode attempts to reproduce the MATLAB intrinsic function mtimes results exactly. When there was a choice between faster code that did not match the MATLAB intrinsic mtimes function results exactly vs slower code that did match the MATLAB intrinsic mtimes function results exactly, the choice was made to use the slower code. Speed improvements were only made in cases that did not cause a mismatch. Caveat: I have only tested on a 32-bit PC with later versions of MATLAB (R2006b - R2009b). This works, but MATLAB may use different algorithms for mtimes in earlier versions or on other machines than I was unable to test, so even this mode may not match the MATLAB intrinsic mtimes function exactly in these other cases. ‘MATLAB’ mode is the default mode when **mtimesx** is first loaded and executed (i.e., the first time you use **mtimesx** in your MATLAB session and the first time you use **mtimesx** after clearing it). You can set this mode for all future calculations with the command **mtimesx**('MATLAB') (case insensitive).

'SPEED' mode

This mode attempts to reproduce the MATLAB intrinsic function mtimes results closely, but not necessarily exactly. When there was a choice between faster code that did not exactly match the MATLAB intrinsic mtimes function vs slower code that did match the MATLAB intrinsic mtimes function, the choice was made to use the faster code. Speed improvements were made in all cases that I could identify, even if they caused a slight mismatch with the MATLAB intrinsic mtimes results. NOTE: The mismatches are the results of doing calculations in a different order and are not indicative of being less accurate. You can set this mode for all future calculations with the command **mtimesx**('SPEED') (case insensitive).

3) Syntax

mtimesx has full support for transpose ('T'), conjugate transpose ('C'), and conjugate ('G') pre-operations on the input variables. The general syntax to perform the operation op(A) * op(B) is (arguments in brackets [ ] are optional):

```
    mtimesx( A [,transa] ,B [,transb] [,mode] )
```

Where:
A = a single, double, or double sparse scalar, vector, matrix, or nD-array
B = a single, double, or double sparse scalar, vector, matrix, or nD-array

And where transa, transb, and mode are the optional inputs:
transa = A character indicating a pre-operation on A:
transb = A character indicating a pre-operation on B:
    The pre-operation can be any of:
    'N' or 'n' = No pre-operation (the default if trans_ is missing)
    'T' or 't' = Transpose
    'C' or 'c' = Conjugate Transpose
    'G' or 'g' = Conjugate (no transpose)
mode = 'MATLAB' or 'SPEED' (case insensitive, sets mode for current and future calculations)

Note: The ‘N’, ‘T’, and ‘C’ have the same meanings as the direct inputs to the BLAS routines. The ‘G’ input has no direct BLAS counterpart, but was relatively easy to implement in **mtimesx** and saves time (as opposed to computing conj(A) or conj(B) explicitly before calling **mtimesx**).

Note: You cannot combine double sparse and single inputs, since MATLAB does not support a single sparse result. You also cannot combine sparse inputs with full nD (n > 2) inputs, since MATLAB does not support a sparse nD result. The only exception is a sparse scalar times an nD full array. In that special case, **mtimesx** will treat the sparse scalar as a full scalar and return a full nD result.

You can also call **mtimesx** specifically to get or set the calculation mode without actually doing any calculation. e.g.,

```
    M = mtimesx( [mode] )
```

Where:
M = a char string giving the mode that was in use before the **mtimesx** call
mode = a char string setting the future calculation mode (optional)

Examples:

```
C = mtimesx(A,B)              % performs the calculation C = A * B
C = mtimesx(A,'T',B)          % performs the calculation C = A.' * B
C = mtimesx(A,B,'g')          % performs the calculation C = A * conj(B)
C = mtimesx(A,'c',B,'C')      % performs the calculation C = A' * B'
mtimesx(‘SPEED’)              % sets future calculations to ‘SPEED’ mode
C = mtimesx(A,'g',B,’c’)      % performs the calculation C = conj(A) * B’
C = mtimesx(A,B,’MATLAB’)     % performs the calculation C = A * B in MATLAB
                              % mode. All future calculations use MATLAB mode too
C = mtimesx(A,'T',B,’G’)      % performs the calculation C = A.' * conj(B)
```

4) Multi-Dimensional Support

mtimesx supports nD inputs. For these cases, the first two dimensions specify the matrix multiply involved. The remaining dimensions are duplicated and specify the number of individual matrix multiplies to perform for the result. i.e., **mtimesx** treats these cases as arrays of 2D matrices and performs the operation on the associated pairings. For example:

If A is (2,3,4,5) and B is (3,6,4,5), then
mtimesx(A,B) would result in C(2,6,4,5), where C(:,:,i,j) = A(:,:,i,j) * B(:,:,i,j), i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:

```
C = zeros(2,6,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = A(:,:,m,n) * B(:,:,m,n);
    end
end
```

The first two dimensions must conform using the standard matrix multiply rules taking the transa and transb pre-operations into account, and dimensions 3:end must match exactly or be singleton (equal to 1). If a dimension is singleton then it is virtually expanded to the required size (i.e., equivalent to a repmat operation to get it to a conforming size but without the actual data copy). This is equivalent to a bsxfun capability for matrix multiplication. For example:

If A is (2,3,4,5) and B is (3,6,1,5), then
mtimesx(A,B) would result in C(2,6,4,5), where C(:,:,i,j) = A(:,:,i,j) * B(:,:,1,j), i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:

```
C = zeros(2,6,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = A(:,:,m,n) * B(:,:,1,n);
    end
end
```

When a transpose (or conjugate transpose) is involved, the first two dimensions are transposed in the multiply as you would expect. For example:

If A is (3,2,4,5) and B is (3,6,4,5), then
mtimesx(A,'C',B,'G') gives C(2,6,4,5), where C(:,:,i,j) = A(:,:,i,j)' * conj(B(:,:,i,j)), i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:

```
C = zeros(2,6,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = A(:,:,m,n)' * conj( B(:,:,m,n) );
    end
end
```

If A is a scalar (1,1) and B is (3,6,4,5), then
mtimesx(A,'G',B,'C') would result in C(6,3,4,5), where C(:,:,i,j) = conj(A) * B(:,:,i,j)', i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:
```
C = zeros(6,3,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = conj(A) * B(:,:,m,n)';
    end
end
```

5) Philosophy

What **mtimesx** is:

5.1) An Improvement on the intrinsic MATLAB mtimes function for speed whenever possible. The amount of improvement you will get with **mtimesx** is highly dependent on your particular computer, C compiler, MATLAB version, and whether you are running **mtimesx** in 'MATLAB' mode or 'SPEED' mode. Older versions of MATLAB will see the most improvement, while later versions of MATLAB may see little to no benefit in many cases. **mtimesx** also has direct support for nD operations, treating nD arrays as arrays of 2D arrays. See the following test run m-files for comprehensive results on your particular setup (CAUTION: These tests can take several hours to run … best to do it overnight:

mtimesx_test_ddequal.m	% A test program for (double) * (double) equality
mtimesx_test_ddspeed.m	% A test program for (double) * (double) speed
mtimesx_test_ssequal.m	% A test program for (single) * (single) equality
mtimesx_test_ssspeed.m	% A test program for (single) * (single) speed
mtimesx_test_dsequal.m	% A test program for (double) * (single) equality
mtimesx_test_dsspeed.m	% A test program for (double) * (single) speed
mtimesx_test_sdequal.m	% A test program for (single) * (double) equality
mtimesx_test_sdspeed.m	% A test program for (single) * (double) speed
mtimesx_test_nd.m	% A test program for multi-dimensional speed and equality

5.2) An example of calling BLAS routines from a C-mex routine (non C programmers need not worry: the routine is self-building ... you don't have to know anything about C or mex to use **mtimesx** ... just skip this section). The source code is generously commented so that you can (hopefully) follow why a certain routine was used, or why custom code was used instead of a BLAS routine. Creating a matrix and scalar multiply routine for the BLAS examples gives the opportunity to provide an example that has a practical use and at the same time allows direct comparison with MATLAB results.

What **mtimesx** is *not*:

5.3) An attempt to reproduce the exact BLAS calling sequences that MATLAB uses. For some of the general matrix-matrix and matrix-vector operations, I would not be surprised to find that **mtimesx** *was* using the exact same BLAS calling sequences, but this is not the goal and is not the claim. Indeed, this would have been a conflict with goal 5.1 above in many cases. Keep this important point in mind ... **mtimesx** will *not* reproduce MATLAB results exactly in many 'SPEED' mode cases because that is not the main goal of that mode. But the **mtimesx** 'SPEED' mode *will* produce a result that is just as accurate as the MATLAB result ... and in some special cases will run many times faster than the MATLAB instinsic function mtimes (not just a few % faster, actually 3x - 4x faster).

6) BLAS Routines Used

The BLAS routines used are DDOT, DGEMV, DGEMM, DSYRK, and DSYR2K for double variables, and SDOT, SGEMV, SGEMM, SSYRK, and SSYR2K for single variables. These routines are (description taken from www.netlib.org):

DDOT and SDOT:

```
*     forms the dot product of two vectors.
```

DGEMV and SGEMV:

```
*  Performs one of the matrix-vector operations
*
*     y := alpha*A*x + beta*y,   or   y := alpha*A'*x + beta*y,
*
*  where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.
```

DGEMM and SGEMM:

```
*  Performs one of the matrix-matrix operations
*
*     C := alpha*op( A )*op( B ) + beta*C,
*
*  where  op( X ) is one of
*
*     op( X ) = X   or   op( X ) = X',
*
*  alpha and beta are scalars, and A, B and C are matrices, with op( A ) an m by k matrix,
*  op( B )  a  k by n matrix and  C an m by n matrix.
```

DSYRK and SSYRK:

```
*  Performs one of the symmetric rank k operations
*
*     C := alpha*A*A' + beta*C,
*
*  or
*
*     C := alpha*A'*A + beta*C,
*
*  where  alpha and beta  are scalars, C is an  n by n  symmetric matrix and  A  is an  n by k
*  matrix in the first case and a  k by n  matrix in the second case.
```

DSYR2K and SSYR2K:

```
*  Pperforms one of the symmetric rank 2k operations
*
*     C := alpha*A*B' + alpha*B*A' + beta*C,
*
*  or
*
*     C := alpha*A'*B + alpha*B'*A + beta*C,
*
*  where  alpha and beta  are scalars, C is an  n by n  symmetric matrix and  A and B  are  n by k
*  matrices  in the  first  case  and  k by n matrices in the second case.
```

7) Supported Operations

At first, my only goal was 5.2 above. However, after some preliminary testing it became apparent that with some custom code to minimize memory access times and take full advantage of symmetric cases there could be some substantial speed improvements by pursuing goal 5.1 above, particularly for complex operations on older versions of MATLAB. Hence you will find extensive custom code for some of the cases, particularly the scalar multiplies.

mtimesx supports the following operations (arguments in either order):

1D vector * scalar
2D matrix * scalar
nD array * scalar
vector * vector (inner or outer product)
matrix * vector
matrix * matrix
nD matrix * nD matrix (treated as arrays of 2D matrices with singleton expansion)

Nearly all operations support nD arrays with the understanding that only the first two dimensions are used in any transpose operations. The main restriction on nD arrays comes from MATLAB itself ... you can't combine sparse and nD (n > 2) variables because MATLAB does not support a sparse nD result.

8) Speed Improvements

It bears repeating that the **mtimesx** timing results are <u>highly</u> dependent on your particular computer, C compiler, and version of MATLAB. **mtimesx** uses custom code in many places instead of BLAS calls in an effort to minimize memory access times. The effectiveness of this custom code can vary quite a bit when compared directly to the intrinsic MATLAB mtimes, particularly with respect to the MATLAB version involved. Nevertheless, here are some sample timings with huge variables (e.g., 100MB) using R2008a with the lcc compiler on 32-bit WinXP:

```
  SPEED MODE
 (% faster mtimesx over mtimes)   Real*Real  Real*Cplx  Cplx*Real  Cplx*Cplx
  conj(Scalar)  * Vector.'              -3%        60%        -2%        60%
  conj(Vector)  * Scalar.'              -4%        -3%        58%        51%
  conj(Array)   * Scalar.'              76%        36%        83%        70%
  conj(Vector)  i Vector.'              -4%         7%       172%       169%
  conj(Vector)  o Vector.'              -8%         6%         6%        26%
  conj(Vector)  * Matrix.'              -0%         0%         0%        -0%
  conj(Matrix)  * Vector.'              -0%        -0%       360%       142%
  conj(Matrix)  * Matrix.'              -0%         0%         3%         2%
```

Many of the operations offer no speed improvement, but some of them offer a substantial improvement. For example, the table above shows that **mtimesx** is 360% faster than MATLAB mtimes for the computation conj(complex matrix) * (real vector).' in 'SPEED' mode. A quick examination of the table reveals that this is not an isolated incident. There are several calculations in the sample table above where **mtimesx** is 10's and 100's percent faster than MATLAB mtimes. Similar speed improvements are obtained with other calculations not shown.

Double sparse matrix operations are supported, but not always directly. For matrix * scalar operations, custom code is used to produce a result that minimizes memory access times. All other operations, such as matrix * vector or matrix * matrix, or any operation involving a transpose or conjugate transpose, are obtained with calls back to the MATLAB intrinsic mtimes function. Thus for most non-scalar sparse operations, **mtimesx** is simply a thin wrapper around the intrinsic MATLAB function mtimes and you will see no speed improvement.

How does **mtimesx** get speed improvements over the MATLAB intrinsic mtimes function?

By reducing memory access times and taking full advantage of conjugate and symmetric cases. The MATLAB intrinsic mtimes function is excellent, but it doesn't optimize memory access for all operation combinations and sometimes uses memory inefficient algorithms to calculate the result. This is particularly true for older versions of MATLAB. For many cases, what I have done is to call a different set of BLAS routines or create custom code. The MATLAB sequence of calling the BLAS routines for some of these operations, particularly for complex operations, involves accessing the input variables twice. Custom code can sometimes avoid this and access the input variables only once, reducing the total memory access time to perform the operation. Some examples:

Example 1: Large complex matrix conjugate transposed times a "thin" complex matrix.

```
>> A=rand(3000)+rand(3000)*1i;
>> B=rand(3000,3)+rand(3000,3)*1i;
>> tic;A' * B;toc
Elapsed time is 0.440585 seconds.
>> tic;mtimesx(A,'c',B,'SPEED');toc
Elapsed time is 0.217146 seconds.
>> isequal(A'*B,mtimesx(A,'c',B,'SPEED'))
ans =
    0
```

So how did **mtimesx** beat mtimes for this operation? Since the first matrix A is transposed, the actual memory access for this matrix can be done by physical columns, not rows, and the physical column elements are contiguous in memory. Custom code was written to take advantage of this fact, and performs the operation as a series of complex dot product type operations on the columns of A and B. The custom code does the real and imaginary part of the dot product result all within the same loop, so the input array contents are accessed only once. The MATLAB intrinsic mtimes apparently uses a series of BLAS calls, forcing the input array memory to be accessed twice. This probably accounts for the 2x speed improvement. Because the underlying calculations are done slightly differently, the answers are slightly different. But the **mtimesx** result is just as accurate as the MATLAB intrinsic mtimes result. **mtimesx** uses a custom C implementation of a basic dot product algorithm using loop unrolling to gain a speed and accuracy improvement. A loop block size of 10 for the unrolling was optimum in testing, so that is what is used in the custom code. Custom dot product code, rather than calling DDOT or SDOT directly, is necessary to avoid accessing the input variables twice.

Example 2: Large complex vector outer product.

```
>> A = rand(3000,1)+rand(3000,1)*1i;
>> B = rand(1,3000)+rand(1,3000)*1i;
>> tic;A*B;toc
Elapsed time is 0.467077 seconds.
>> tic;mtimesx(A,B,'SPEED');toc
Elapsed time is 0.249749 seconds.
>> isequal(A*B,mtimesx(A,B))  % 'SPEED' option not needed since it carries over from the
previous call
ans =
    1
```

Again, the speed improvement is the result of custom code in **mtimesx** that avoids accessing the input variables twice. Here the results are exactly the same.

Example 3: Large complex sparse matrix times a complex scalar.

```
>> A = sprand(10000,10000,.1);
>> A = A + A*1i;
>> B = rand + rand*1i;
>> mtimesx('SPEED');  % This setting will carry forward to all subsequent calls
>> tic;conj(a)*b;toc
Elapsed time is 7.740495 seconds.
>> tic;mtimesx(a,'G',b);toc
Elapsed time is 0.520425 seconds.
>> isequal(conj(A)*B,mtimesx(A,'G',B))
ans =
    1
```

The dramatic speed improvement here is because **mtimesx** treats the operation as a scalar times a 1D array. No special sparse matrix multiply code is needed. Apparently the MATLAB intrinsic mtimes function calls special sparse matrix multiply algorithms for this. Here the results are exactly the same. I will quickly point out that this example is for an older version of MATLAB. The latest version of MATLAB (R2009b) has improved this calculation but even in this case **mtimesx** is nearly 400% faster (as compared to nearly 1500% faster in the above example).

9) List of Included Files

List of files:

mtimesx.m	% A short m-file, mainly used for the help text
mtimesx_build.m	% The file used to build mtimesx.mexext for PC Windows
mtimesx_sparse.m	% A short m-file used for sparse operations
mtimesx_test_ddequal.m	% A test program for (double) * (double) equality
mtimesx_test_ddspeed.m	% A test program for (double) * (double) speed
mtimesx_test_ssequal.m	% A test program for (single) * (single) equality
mtimesx_test_ssspeed.m	% A test program for (single) * (single) speed
mtimesx_test_dsequal.m	% A test program for (double) * (single) equality
mtimesx_test_dsspeed.m	% A test program for (double) * (single) speed
mtimesx_test_sdequal.m	% A test program for (single) * (double) equality
mtimesx_test_sdspeed.m	% A test program for (single) * (double) speed
mtimesx_test_nd.m	% A test program for multi-dimensional speed and equality
mtimesx.c	% C source code for mexFunction interface
mtimesx_RealTimesReal.c	% The C source code for the actual multiplication
mtimesx_20100106.pdf	% The file you are currently reading

For PC Windows, **mtimesx** attempts to be self-building. When you first run **mtimesx**, the file that will execute will be mtimesx.m. If this file executes, then that means that the mex dll routine has not been built yet. So the only code in mtimesx.m is a call to mtimesx_build.m to build the mex dll routine and then call it. mtimesx_build will try to autodetect if the machine is PC Windows. If it is, it will try to self-build the mex dll routine. If not, it will exit with brief instructions on how to manually compile the mex dll routine.

Once the dll building is complete, you will also have an additional file:

mtimesx.mexext	% The actual mtimesx function

From now on, whenever you invoke **mtimesx** it is this file that actually executes. The mexext is replaced with the actual mex dll extension for your particular computer and operating system. For example, on 32-bit windows the extension is mexw32 for later versions of MATLAB.

Sparse matrix multiply operations and transpose operations are not directly supported in the C code. Instead, a call-back to MATLAB is done to perform these using the mtimesx_sparse.m function. So you will not see any difference between **mtimesx** and the MATLAB mtimes function for these cases. Where you *will* see a difference is in the op(scalar) * op(sparse) cases, where custom code is used to minimize memory access in **mtimesx**. These cases can yield a significant speed improvement over the MATLAB mtimes function, particularly for older versions of MATLAB.

10) Testing

CAUTION: The only tested configurations for **mtimesx** are PC 32-bit Windows XP with various MATLAB versions R2006b - R2009b and the lcc compiler and Microsoft Visual C/C++ 8 (2005) compiler. All other configurations are untested. The author would like to solicit help from the community in getting **mtimesx** (and the self-building code in mtimesx_build.m) to work under other configurations.

11) Upgrades

Future upgrades planned:

- Bug fixes, of course.
- More extensive test routines.
- Expanded documentation and examples.
- Support for 64-bit and linux and mac machines, particularly for self building. But this will depend on other users willing to supply this code to me, since I do not have access to these machines for development or testing.

Future upgrades that I *may* consider depending on my time availability and the popularity of **mtimesx**:

- More speed improvements to the current code if I can come up with better algorithms. Does anybody out there have a fast matrix transpose algorithm? Please contact me if you do.
- Custom code for (single * double) and (double * single) multiplies. As currently written, **mtimesx** simply does a conversion of the input argument(s) first and then does the multiply. This can run quite a bit slower than the MATLAB intrinsic mtimes function.
- Custom code for sparse transpose and conjugate transpose operations. The goal here again would be to see if a speed improvement can be achieved. Implementation will depend on my ability to find an efficient sparse matrix transpose algorithm.
- Support for older versions of MATLAB, particularly versions prior to 7. My guess is that some of the MATLAB code and/or API functions I used might not be compatible. Again, this will depend on other users willing to help modify the code, since I do not have access to these older versions of MATLAB for development or testing.

12) Contact the Author

Feel free to post items of general interest to other users (bug reports, performance data, questions about usage or optimizations, etc) directly on the FEX of course. But if you have modified the code for your version of MATLAB (older version, non-PC machine, non-supported C compiler, etc.) please feel free to contact me directly and I will try to incorporate them into future **mtimesx** upgrades. You can reach me at

a#lassyguy%ho$mail_com (replace # with k, % with @, $ with t, _ with .)

James Tursa